Lecture 08:
Logical Operations

Logical Operators (1/3)

- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output for each. E.g.,

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logical Operators (2/3)

- Logical Instruction Syntax:
  1   2,3,4

  - where
    1) operation name
    2) register that will receive value
    3) first operand (register)
    4) second operand (register) or immediate (numerical constant)

  - In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

Logical Operators (3/3)

- Instruction Names:
  - and, or: Both of these expect the third argument to be a register
  - andi, ori: Both of these expect the third argument to be an immediate

- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0’s of the inputs, bit 1 by the bit 1’s, etc.
  - C: Bitwise AND is & (e.g., z = x & y;)
  - C: Bitwise OR is | (e.g., z = x | y;)

Uses for Logical Operators (1/3)

- Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.
- This can be used to create a mask.
  - Example:
    1011 0110 1010 0100 0011 1101 1001 1010
    mask: 0000 0000 0000 0000 0000 1111 1111 1111
  - The result of anding these:
    0000 0000 0000 0000 0000 1101 1001 1010
    mask last 12 bits
Uses for Logical Operators (2/3)

• The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).

• Thus, the and operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
  • In particular, if the first bitstring in the above example were in $t0$, then the following instruction would mask it:
    ```
    and $t0, $t0, 0xFFF
    ```

Uses for Logical Operators (3/3)

• Similarly, note that or’ing a bit with 1 produces a 1 at the output while or’ing a bit with 0 produces the original bit.

• This can be used to force certain bits of a string to 1s.
  • For example, if $t0$ contains 0x12345678, then after this instruction:
    ```
    or $t0, $t0, 0xFFFF
    ```
  • ...$t0$ contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

Shift Instructions (1/4)

• Move (shift) all the bits in a word to the left or right by a number of bits.
  • Example: shift right by 8 bits
    ```
    0001 0010 0011 0100 0101 0110 0111 1000
    0000 0000 0001 0010 0011 0100 0101 0110
    ```
  • Example: shift left by 8 bits
    ```
    0001 0010 0011 0100 0101 0110 0111 1000
    0011 0100 0101 0110 0111 1000 0000 0000
    ```

Shift Instructions (2/4)

• Shift Instruction Syntax:
  1  2,3,4
  • where
    1) operation name
    2) register that will receive value
    3) first operand (register)
    4) shift amount (constant < 32)

• MIPS shift instructions:
  1. **sll** (shift left logical): shifts left and fills emptied bits with 0s
  2. **srl** (shift right logical): shifts right and fills emptied bits with 0s
  3. **sra** (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (3/4)

• Example: shift right arith by 8 bits
  ```
  0001 0010 0011 0100 0101 0110 0111 1000
  0000 0000
  ```
  • Example: shift right arith by 8 bits
    ```
    1001 0010 0011 0100 0101 0110 0111 1000
    1111 1111 1001 0010 0011 0100 0101 0110
    ```

Shift Instructions (4/4)

• Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:
  ```
  a *= 8; (in C)
  ```
  would compile to:
  ```
  sll $s0,$s0,3 (in MIPS)
  ```
• Likewise, shift right to divide by powers of 2
  • remember to use sra
Example: Uses for Shift Instructions (1/4)

• Suppose we want to extract byte 0 (rightmost 8 bits) of a word in $t0.
  Simply use:
  ```
  andi $t0,$t0,0xFF
  ```
• Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word in $t0. We can use:
  ```
  andi $t0,$t0,0xFF00
  ```
  but then we still need to shift to the right by 8 bits...

Example: Uses for Shift Instructions (2/4)

• Could use instead:
  ```
  sll $t0,$t0,16
  srl $t0,$t0,24
  ```

Example: Uses for Shift Instructions (3/4)

• In decimal:
  - Multiplying by 10 is same as shifting left by 1:
    ```
    714_{10} \times 10_{10} = 7140_{10}
    56_{10} \times 10_{10} = 560_{10}
    ```
  - Multiplying by 100 is same as shifting left by 2:
    ```
    714_{10} \times 100_{10} = 71400_{10}
    56_{10} \times 100_{10} = 5600_{10}
    ```
  - Multiplying by 10^n is same as shifting left by n

Example: Uses for Shift Instructions (4/4)

• In binary:
  - Multiplying by 2 is same as shifting left by 1:
    ```
    11_{2} \times 10_{2} = 110_{2}
    1010_{2} \times 10_{2} = 10100_{2}
    ```
  - Multiplying by 4 is same as shifting left by 2:
    ```
    11_{2} \times 100_{2} = 1100_{2}
    1010_{2} \times 100_{2} = 101000_{2}
    ```
  - Multiplying by 2^n is same as shifting left by n

“And in Conclusion…”

• Logical and Shift Instructions
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, **sll**, for multiplication by powers of 2
  - Use shift right arithmetic, **sra**, for division by powers of 2.
• New Instructions:
  ```
  and, andi, or, ori, sll, srl, sra
  ```